Barriers

- Barriers are synchronization primitives that ensure that some processes do not outrun others – if a process reaches a barrier, it has to wait until every process reaches the barrier

- When a process reaches a barrier, it acquires a lock and increments a counter that tracks the number of processes that have reached the barrier – it then spins on a value that gets set by the last arriving process

- Must also make sure that every process leaves the spinning state before one of the processes reaches the next barrier

Barrier Implementation

```
LOCK(bar.lock);
if (bar.counter == 0)
  bar.flag = 0;
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = 1;
}
else
  while (bar.flag == 0)  { };
```

Sense-Reversing Barrier Implementation

```
local_sense = !(local_sense);
LOCK(bar.lock);
mycount = bar.counter++;
UNLOCK(bar.lock);
if (mycount == p) {
  bar.counter = 0;
  bar.flag = local_sense;
}
else {
  while (bar.flag != local_sense)  { };
}
```

Review: ensuring atomicity via locks

```
void deposit(Acct account, int amount)
{
    lock(account.lock);
    int tmp = bank.get(account);
    tmp += amount;
    bank.put(account, tmp);
    unlock(account.lock);
}
```

- `Deposit` is a read-modify-write operation: want "deposit" to be atomic with respect to other bank operations on this account
- Locks are one mechanism to synchronize threads to ensure atomicity of update (via ensuring mutual exclusion on the account)

Programming with transactions

```
void deposit(Acct account, int amount)
{
   lock(account.lock);
   int tmp = bank.get(account);
   tmp += amount;
   bank.put(account, tmp);
   unlock(account.lock);
}
```

```
void deposit(Acct account, int amount)
{
   atomic {
     int tmp = bank.get(account);
     tmp += amount;
     bank.put(account, tmp);
   }
}
```

- **Atomic construct is declarative**
  - Programmer states what to do (maintain atomicity of this code), not how to do it
  - No explicit use or management of locks

- **System implements synchronization as necessary to ensure atomicity**
  - System could implement atomic { } using a lock
  - Implementation discussed today uses optimistic concurrency: serialization only in situations of true contention (R-W or W-W conflicts)

Declarative vs. imperative abstractions

- Declarative: programmer defines what should be done
  - Execute all these independent 1000 tasks
  - Perform this set of operations atomically

- Imperative: programmer states how it should be done
  - Spawn N worker threads. Assign work to threads by removing work from a shared task queue
  - Acquire a lock, perform operations, release the lock

Transactional Memory (TM)

- **Memory transaction**
  - An atomic and isolated sequence of memory accesses
  - Inspired by database transactions
- **Atomicity (all or nothing)**
  - Upon transaction commit, all memory writes in transaction take effect at once
  - On transaction abort, none of the writes appear to take effect (as if transaction never happened)
- **Isolation**
  - No other processor can observe writes before transaction commits
- **Serializability**
  - Transactions appear to commit in a single serial order
  - But the exact order of commits is not guaranteed by semantics of transaction

Transactional Memory (TM)

- In other words… many of the properties we maintained for a single address in a coherent memory system, we'd like to maintain for sets of reads and writes in a transaction.

Transaction:

Reads: X, Y, Z

Writes: A, X

These memory transactions will either all be observed by other processors, or none of them will. (the effectively all happen at the same time)

Load-linked, store conditional (LL/SC)

- LL/SC is a lite version of transactional memory
- Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)
  - load_linked(x): load value from address
  - store_conditional(x, value): store value to x, if x hasn't been written to since corresponding LL
- Corresponding ARM instructions: LDREX and STREX
- How might LL/SC be implemented on a cache coherent processor?

Motivating transactional memory

Example

Producer-consumer relationships – producers place tasks at the tail of a work-queue and consumers pull tasks out of the head

```
Enqueue
  transaction begin
    if (tail == NULL)
      update head and tail
    else
      update tail
  transaction end
```

```
Dequeue
    transaction begin
      if (head->next == NULL)
        update head and tail
      else
        update head
    transaction end
```

With locks, neither thread can proceed in parallel since head/tail may be updated – with transactions, enqueue and dequeue can proceed in parallel – transactions will be aborted only if the queue is nearly empty

Advantages (promise) of transactional memory

- **Easy to use synchronization construct**
  - It is difficult for programmers to get synchronization right
  - Programmer declares need for atomicity, system implements it well
  - Claim: transactions are as easy to use as coarse-grain locks
- **Often performs as well as fine-grained locks**
  - Provides automatic read-read concurrency and fine-grained concurrency
  - Performance portability: locking scheme for four CPUs may not be the best scheme for 64 CPUs
  - Productivity argument for transactional memory: system support for transactions can achieve 90% of the benefit of expert programming with fined-grained locks, with 10% of the development time
- **Failure atomicity and recovery**
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- **Composability**
  - Safe and scalable composition of software modules

Performance: locks vs. transactions

"TCC" is a TM system implemented in hardware